# STAT 378 Assignment 4 Final Project

*Siwei Li Siqi Wang Ruhan Zhang*

*November 6, 2015*

### Problem 1. Metropolis-Hastings Algorithm

Step 1: Main Algorithmm for Beta distribution-Metropolis Hasting
Here is the algorithm scheme:

1. Generate initial value $x_0$ from $Uniform(0,1)$ and set the iteration number $T$ and number $c$.
2. Set index $t = 1$.
3. Generate random draws $(x_t)$ where
   - Generate $x^*$ from distribution $Beta(cx_{t-1}, c(1-x_{t-1}))$
   - Calculate proposal ratio $P_{prop} = \frac{Q(x_{t-1}|x^*)}{Q(x^*|x_{t-1})}$ where $Q(x_1|x_2)$ follows the distribution $Beta(cx_2, c(1-x_2))$
   - Calcaulate posterior ratio $P_{pos} = \frac{f(x^*)}{f(x_{t-1})}$ where $f$ is the density of $Beta(\alpha, \beta)$
   - Calculate acceptance ratio $P = P_{prop} * P_{pos}$
   - Generate $u$ from $Uniform(0,1)$
   - If $u < min(1, P)$, $x_t = x^*$; Otherwise $x_t = x_{t-1}$
4. Record sample $x_t$
5. Set $t = t + 1$, return to step 3 if $t \leqslant T$.
6. Return $T$ samples.

```r
set.seed(99)
#prepare for the traceplot in later stages:
library(coda)
```

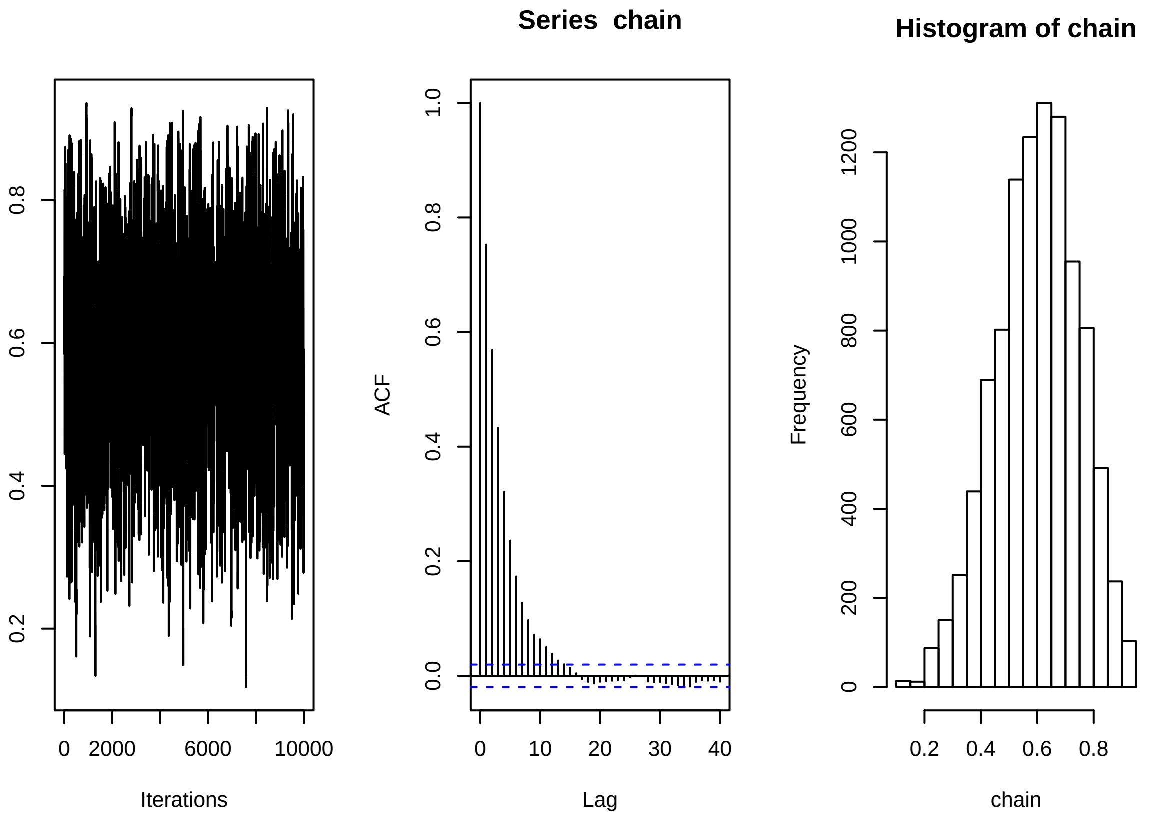
```
## Warning: package 'coda' was built under R version 3.1.3
```

```r
#Metropolis Hasting Procedure:
Beta_Metropolis <- function(alpha=6,beta=4,c,iterations){
  chain <- numeric(iterations+1)
  start_value <- runif(1)
  chain[1] <- start_value
  for(i in 1:iterations){
    current_value <- chain[i]
    #proposal function:
    newbeta <- 1
    while (newbeta == 0 | newbeta == 1) {
      newbeta <- rbeta(1, c*current_value, c*(1-current_value))
    }
    #proposal ratio:
    proposal_ratio <- dbeta(current_value, c*newbeta, c*(1-newbeta)) / dbeta(newbeta, c*current_value, 
    #posterior ratio:
```

```
    posterior_ratio <- dbeta(newbeta, alpha, beta) / dbeta(current_value, alpha, beta)
    #acceptance ratio:
    if(runif(1) < min(1, posterior_ratio * proposal_ratio)){
      chain[i+1] <- newbeta
    }else{
      chain[i+1] <- current_value
    }
  }
  return(chain)
}
```

Step2: Evaluate the performance of the sampler

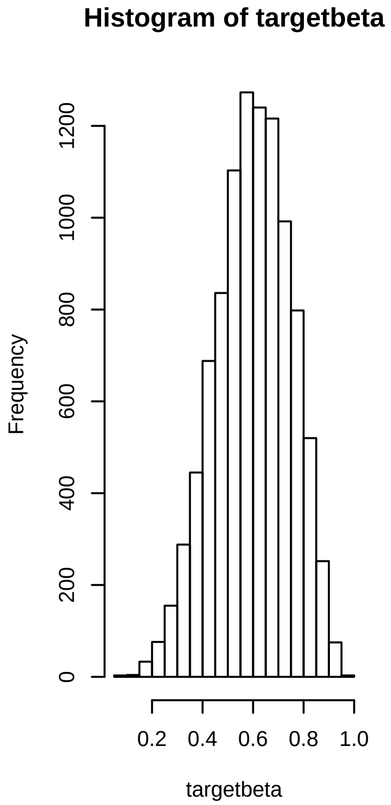
```
chain <- Beta_Metropolis(alpha=6,beta=4,c=1,iterations=10000)
par(mfrow=c(1,3))  #1 row, 3 columns
traceplot(as.mcmc(chain)); acf(chain); hist(chain)  #plot commands
```



```
#graphical comparison
targetbeta <- rbeta(10000,6,4)
hist(targetbeta)
#numerical comparison - Kolmogorov-Smirnov statistic
ks.test(chain,targetbeta)
```

```
## Warning in ks.test(chain, targetbeta): p-value will be approximate in the
## presence of ties
```

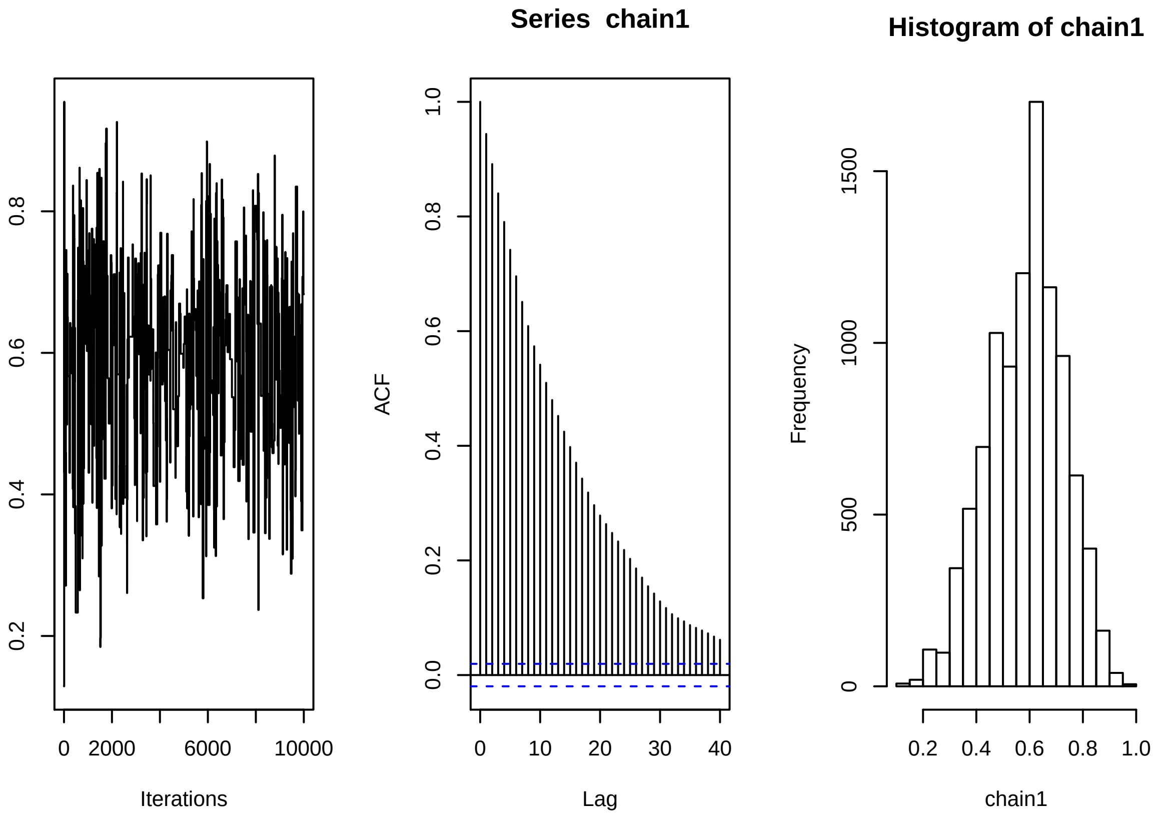
```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  chain and targetbeta
## D = 0.0193, p-value = 0.04914
## alternative hypothesis: two-sided
```



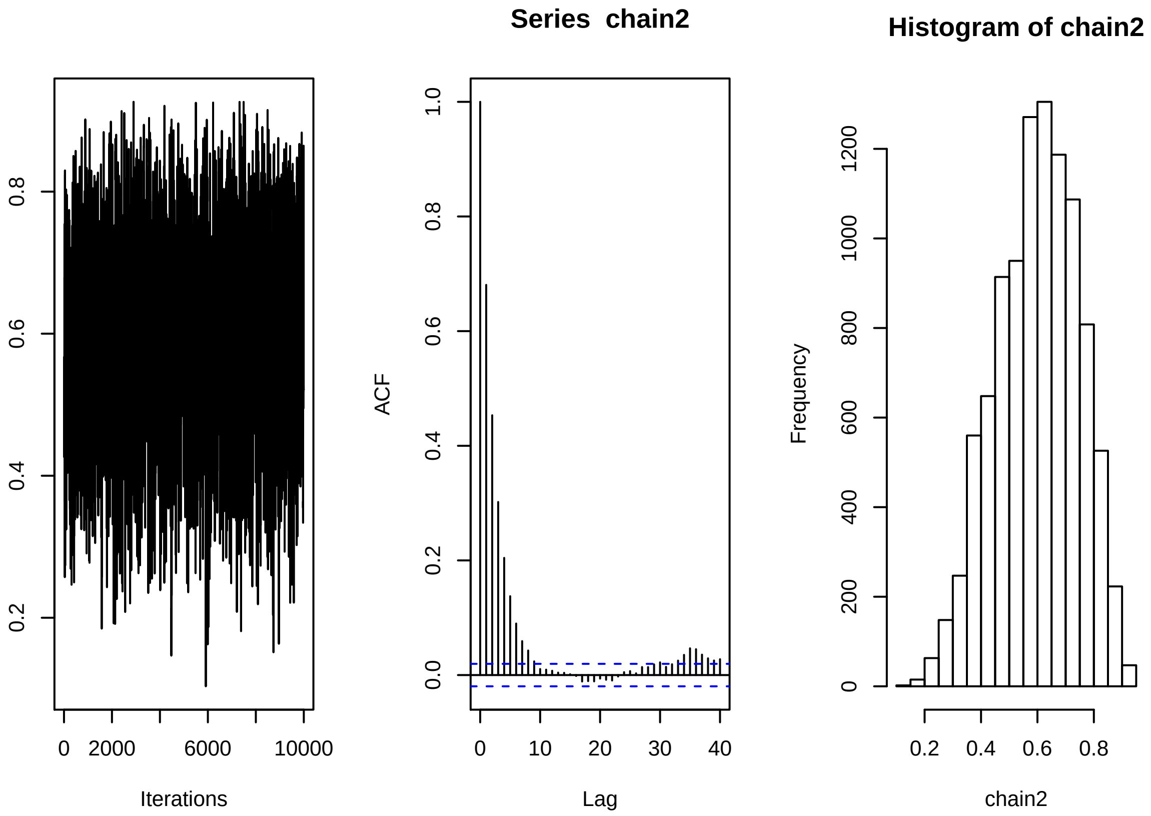
The traceplot shows the value of $x_t$ changes all the time. It indicates that we are actually sampling data with different x-values which implies that our algorithm is making valid approximation.
The autocorrelation plot shows that after a certain number of iterations, the sample will not be affected by the correlation problem, which is a good sign since our aim is to get approximate samples from the target distribution.
The histogram shows the shape of the sample distribution. It seems like a shape from the corresponding beta distribution.
Using Kolmogorov-Smirnov test, we are trying to test that if our chain follows our target Beta distribution, $Beta(\alpha = 6, \beta = 4)$. The null hypothesis of this test states that our chain follows the target Beta distribution. The p-value for this test is 0.04914<0.05 if we are evaluating at 95% significance level. Therefore, we do not reject the null hypothesis. Our metropolis hasting performs quite good in getting samples from the target Beta distribution.

Step3: re-run the sampler with c=0.1, c=2.5 and c=10

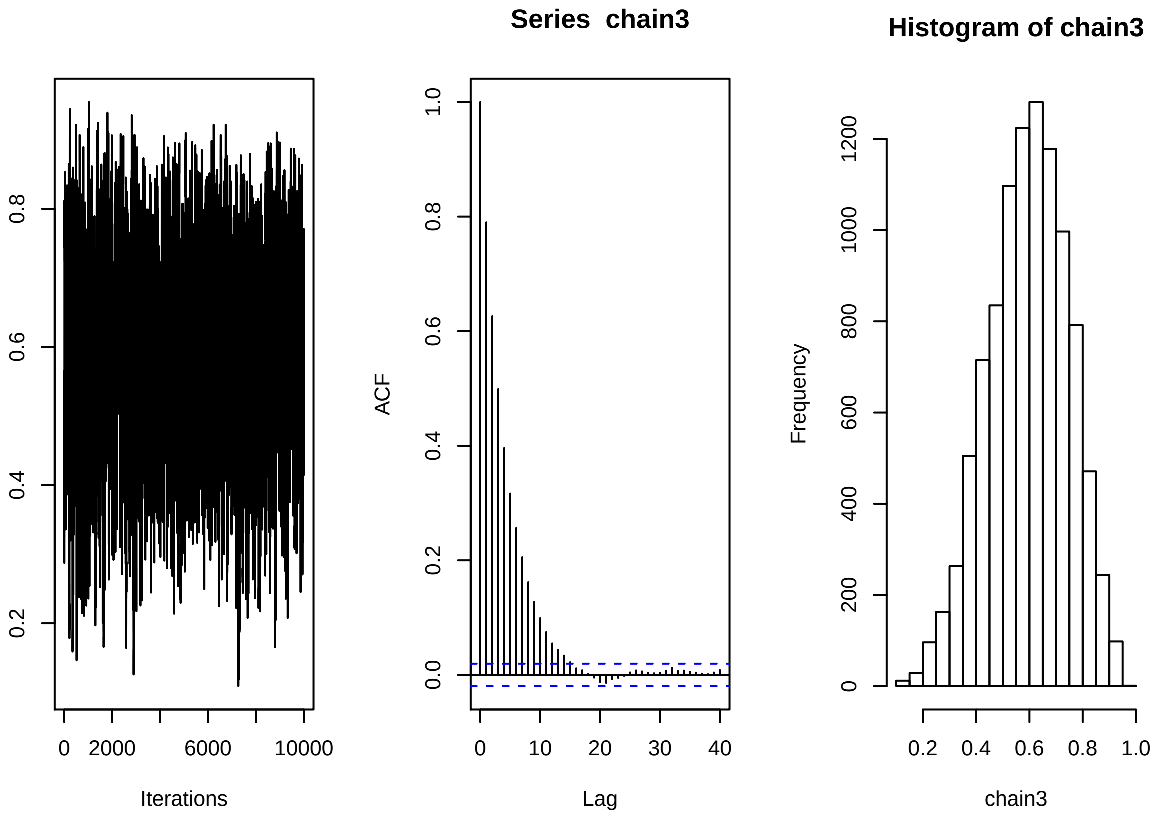
```
par(mfrow=c(1,3))
chain1 <- Beta_Metropolis(alpha=6,beta=4,c=0.1,iterations=10000)
traceplot(as.mcmc(chain1)); acf(chain1); hist(chain1)
```



```
chain2 <- Beta_Metropolis(alpha=6,beta=4,c=2.5,iterations=10000)
traceplot(as.mcmc(chain2)); acf(chain2); hist(chain2)
```



```
chain3 <- Beta_Metropolis(alpha=6,beta=4,c=10,iterations=10000)
traceplot(as.mcmc(chain3)); acf(chain3); hist(chain3)
```



According to the acf plot, the correlation problem solved faster for $c \geqslant 2.5$. The acf is still too large after lag 40 for $c = 0.1$. This penomena indicates that the Markov Chain would converge faster as c increases. Thus it is the best to use $c = 10$.

### Problem 2. Gibbs Sampling

To sample from the full conditional distribution, we need to figure out the full conditional density function first. The problem indicates that

$$p(x|y) \propto y\mathrm{e}^{-yx} \quad 0 < x < B < \infty$$

$$p(y|x) \propto x\mathrm{e}^{-yx} \quad 0 < y < B < \infty$$

Suppose the normalizing constant of $p(x|y)$ is $c$, we can figure out $c$ by

$$\begin{aligned} \int_0^B cy\mathrm{e}^{-yx}dx = 1 \\ c(-\mathrm{e}^{-yx}|_0^B) = 1 \\ c(1 - \mathrm{e}^{-By}) = 1 \end{aligned}$$

So now we know that $c = \frac{1}{1-\mathrm{e}^{-By}}$. Similarly, we can find that the normalizing constant for $p(y|x)$ is $\frac{1}{1-\mathrm{e}^{-Bx}}$. And we can write the density function as

$$p(x|y) = \frac{y\mathrm{e}^{-yx}}{1 - \mathrm{e}^{-By}} \quad 0 < x < B < \infty$$

$$p(y|x) = \frac{x\mathrm{e}^{-yx}}{1-\mathrm{e}^{-Bx}} \quad 0 < y < B < \infty$$

Then the cumulative density functions are

$$F_{X|Y}(x|y) = \frac{1-\mathrm{e}^{-yx}}{1-\mathrm{e}^{-By}} \quad 0 < x < B$$

$$F_{Y|X}(y|x) = \frac{1-\mathrm{e}^{-yx}}{1-\mathrm{e}^{-Bx}} \quad 0 < y < B$$

With some computations, we may obtain the inverse function of two CDFs:

$$F_{X|Y}^{(-1)}(u|x,y) = \frac{-\log\left(1-u+u\mathrm{e}^{-By}\right)}{y} \quad 0 < x < B$$

$$F_{Y|X}^{(-1)}(u|x,y) = \frac{-\log\left(1-u+u\mathrm{e}^{-Bx}\right)}{x} \quad 0 < y < B$$

where $0 \leqslant u \leqslant 1$. Now, we can list the algorithm scheme:

1. Select initial values $(x_0, y_0)$ and set the iteration number $T$.
2. Set index $t = 1$.
3. Generate random draws $(x_t, y_t)$ where
   - Generate two numbers $u_1$, $u_2$ from Unif(0,1)
   - Sample $x_t$ from distribution $F_{X|Y}^{(-1)}(u_1|x_{t-1}, y_{t-1})$
   - Sample $y_t$ from distribution $F_{Y|X}^{(-1)}(u_1|x_t, y_{t-1})$
4. Record sample $(x_t, y_t)$
5. Set $t = t + 1$, return to step 3 if $t \leqslant T$.
6. Return $T$ samples.

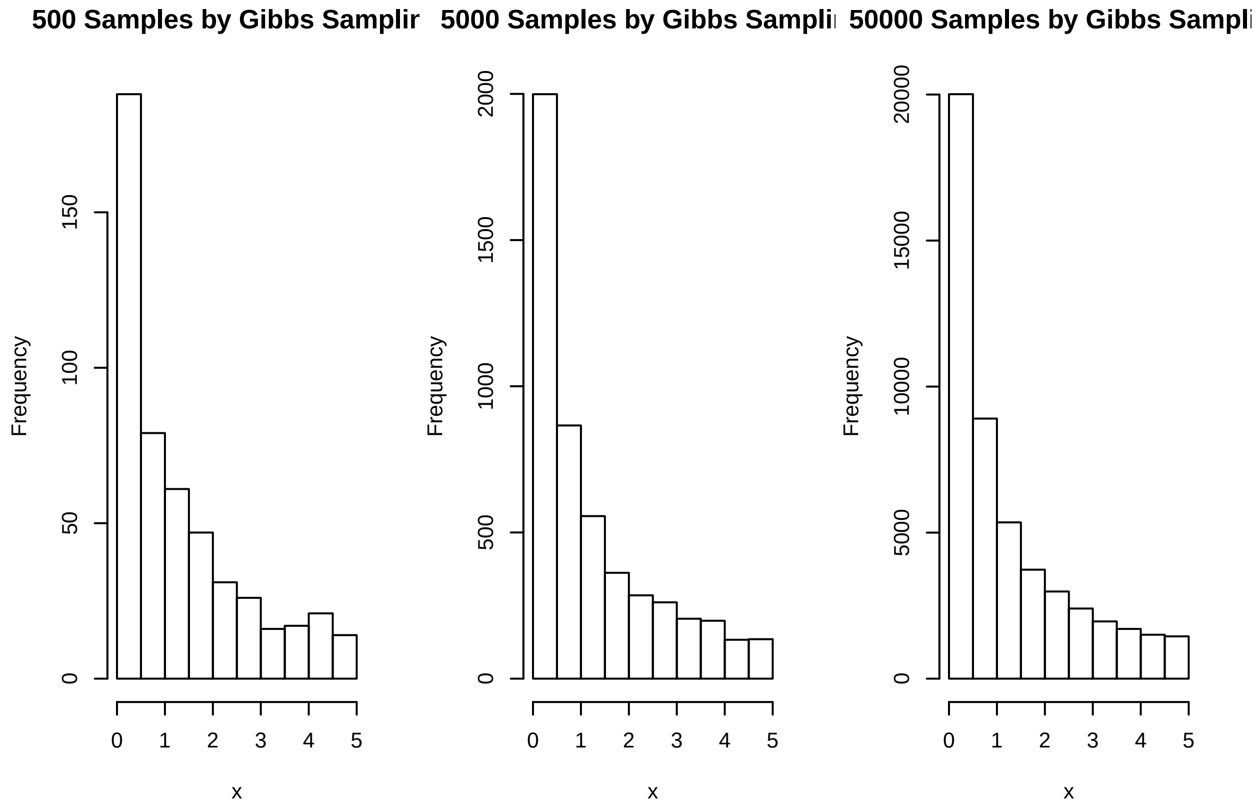
The following is the R code of the algorithm, we denote the Gibbs sampling function as Gibbs.sim():

```r
Gibbs.sim <- function(x0,y0,B,Ite){#The inputs are initial values x0, y0, the bound B and the iteration
  if (x0<0 || x0>B ||y0<0 || y0>B ) {#Check the validity of initial values
    stop("The initial value is not in the correct range.")
  }
  else {x_result=c()
        y_result=c()
        xt=x0
        yt=y0
        for(t in 1:Ite){
          u1=runif(1)
          u2=runif(1)
          xt=-log(1-u1*(1-exp(-B*yt)))/yt
          yt=-log(1-u2*(1-exp(-B*xt)))/xt #the xt here is the updated why, thus the algorithm construct
          x_result=c(x_result, xt)
          y_result=c(y_result, yt)
  }
  return(cbind(x_result,y_result)) #We will do burn in later
  }
}
```

We do not let the function automatically do the burn-in since we think it is not proper to discard the sampling information easily. Besides, the burn-in process is not difficult to be implemented in this case. Now for $B = 5$, we made three samples with sizes $T = 500, 5000, 50000$ and plot the histograms. For each sample, we burn-in the first 200 samples. In order to ensure the correct sample amount, we acutually did 700, 5200, 50200 iterations each time.

```
set.seed(123)
sample500=Gibbs.sim(1,2,5,700)[-(1:200),] #We burn-in the first 200 samples in order to get 500 samples
sample5000=Gibbs.sim(1,2,5,5200)[-(1:200),] #Also burn-in the first 200 samples
sample50000=Gibbs.sim(1,2,5,50200)[-(1:200),] #50000 samples after burn-in

### Plot the histogram ###
par(mfrow=c(1,3))
hist(sample500[,1],xlab="x", main="500 Samples by Gibbs Sampling")
hist(sample5000[,1],xlab="x", main="5000 Samples by Gibbs Sampling")
hist(sample50000[,1],xlab="x", main="50000 Samples by Gibbs Sampling")
```



We can see that all the histograms are skewed to the right. The shape of each plot is similar.
By Gibbs sampling, once the Markov Chain contructed by the algorithm converges to the target distribution which we want to find, we can treat samples as approximates from the target distribution. Thus we can use the mean of samples of $x$ to estimate $E_{p(X)}[X]$. Here are the estimates from three sizes samples:

```
mean(sample500[,1])
```

```
## [1] 1.334719
```

```r
mean(sample5000[,1])
```

```
## [1] 1.265094
```

```r
mean(sample50000[,1])
```

```
## [1] 1.258227
```

The mean of the sample with $T = 500$ is about 1.33 and the other two means are approximately 1.25 and 1.27. We can see that mean estimates of samples with $T = 5000$ and $T = 50000$ are much closer. It is a reasonable phenomena since as the iteration number increases, we will get a more accurate approximation from the target distribution. We may guess that the true mean is around 1.27.

### Problem 3. K-Means

Before starting this part, we first load two packages: fpc and flexclust. The package fpc will allow us to use the command plotcluster() to visualize the clustering result. The package flexclust will allow us to use the function randIndex() to evaluate the clustering result.

```r
library('fpc')
```

```
## Warning: package 'fpc' was built under R version 3.1.3
```

```r
library('flexclust')
```

```
## Warning: package 'flexclust' was built under R version 3.1.3

## Loading required package: grid
## Loading required package: lattice
## Loading required package: modeltools

## Warning: package 'modeltools' was built under R version 3.1.3

## Loading required package: stats4
```

Instead of using the built-in function kmeans() directly, we try to write a R function for K means algorithm. The function is called clusterByKmeans() with two parameters: the data set and the number of groups. We closely follow the K-means algorithm to write this function. The logic of the algorithm as well as the function clusterByKmeans() is as follows.

Suppose we want to cluster all the data points into k groups. Step 1: Randomly pick k data points as the initial centers (means) of the k groups. Step 2: Assign each data point to the group whose center is closest to to this data point in terms of Euclidean distance. Each point is assigned to only one group. This step will then gives k groups of data points. Step 3: Calculate the new centers (means) of the k groups and update the values of the centers (means) of the k groups. Step 4: Repeat Step 2 and Step 3, until the assignment of all data points does not change any more.

The code of the function clusterByKmeans is given below:

```r
clusterByKmeans <- function(dataset, groupNo){

# Obtain the number of rows and the number of columns of the dataset
rowNo <- nrow(dataset)
colNo <- ncol(dataset)

# Create a vector to store the clustering result
# (indicating which point belongs to which cluster)
cluster <- vector("numeric", length=rowNo)

# Create a vector to store the distance between each data point
# and the center of the cluster to which that data point belongs.
dtc <- vector("numeric", length=rowNo)

# Create a matrix to store the centers of the groups
# Use a matrix because each center is a row vector
center <- matrix(0,nrow=groupNo,ncol=colNo)

# Randomly select groupNo points (rows of dataset)
# as the initial centers of the groupNo groups
randomInt <- as.vector(sample(1:rowNo,size=groupNo))
for (i in 1:groupNo){
    center[i,] <- dataset[randomInt[i],]
    center <- matrix(center,groupNo,colNo)
}

# We specify an intial way of clustering:
# all the rest of the points belong to the first cluster,
# while the points chosen as the initial centers belong to its own cluster.
for (i in 1:rowNo){
    cluster[i] <- 1
}
for (j in 1:groupNo){
    cluster[randomInt[j]] <- j
}


# Calculate the initial value of the distance vector, dtc
for (i in 1:rowNo){
    dtc[i] <- sqrt(sum((dataset[i,]-center[cluster[i],])^2))
}

# Create a variable, changed, which indicates
# whether the clustering result is changed in an iteration.
changed <- TRUE

# If the assignment is changed in a iteration, we continue;
# if the assignment no longer changes, we stop.
while(changed==TRUE){

initialCluster <- cluster

for (i in 1:rowNo){
    initialdtc <- dtc[i]
    preCluster <- cluster[i]

    # We shall find a better cluster solution for the point i
    # in terms of smaller Euclidean distance.
    for (j in 1:groupNo){
        # Compute the distance between the point i and the center of the cluster j
        currentdtc <- sqrt(sum((dataset[i,]-center[j,])^2))
        if (currentdtc<initialdtc){
            # Update the clustering result
            cluster[i] <- j
            # Update the distance vector
            dtc[i] <- currentdtc
        }
    }
  }

# Compare the new assignment with the previous assignment.
# The overall assignment is considered as "changed" if the assignment of
# at least one point is changed.
for (i in 1:rowNo){
    if (cluster[i] != initialCluster[i]){
        changed <- TRUE
        break
    }else{
        changed <- FALSE
    }
}

# Then we update the centers of each cluster under the new assignment.
for (k in 1:groupNo){
        # Obtain all the points in the cluster k
        pointsIn <- dataset[cluster==k,]
        pointsMat <- as.matrix(pointsIn)
        # If there is at least one points in the cluster k, we update the center
        ## of the cluster k; otherwise, the center remains unchanged.
        if (nrow(pointsMat)>0){
            center[k,] <- colMeans(pointsMat)
        }else{
            center[k,] <- center[k,]
        }
     }
 }
# return the clustering result
return(cluster)
}
```

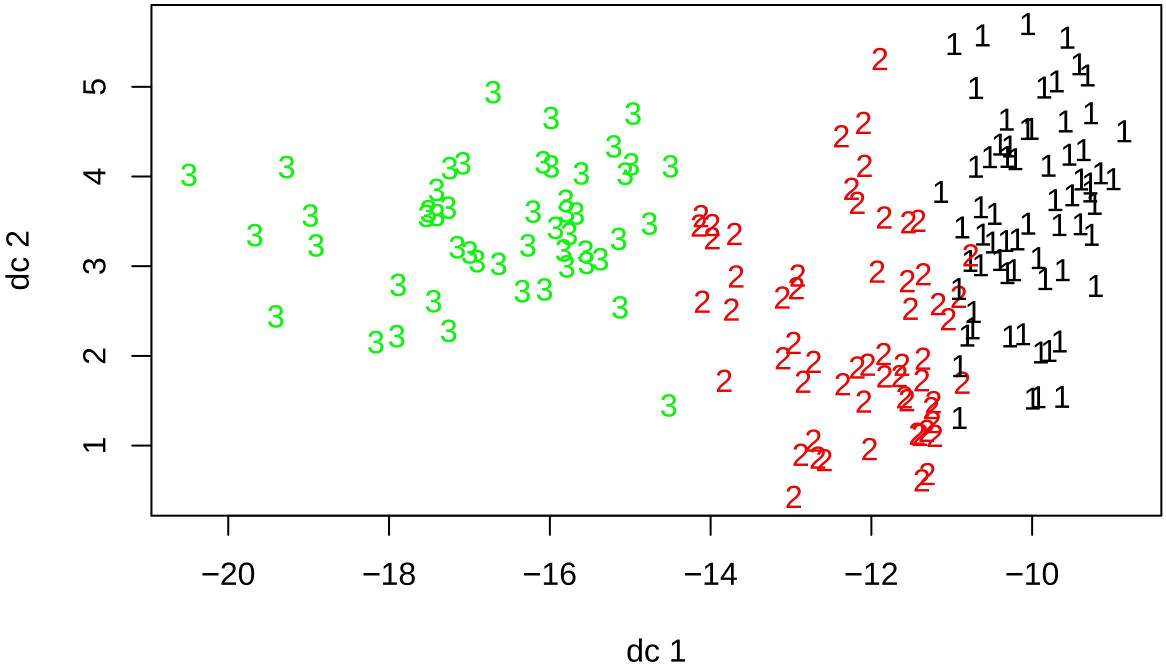
Now we shall use the function clusterByKmeans() to cluster the wine data and the iris data.

K means for the wine data without scaling:

```
# Obtain the wine data from the package "rattle"
data(wine, package="rattle")
head(wine)
```

```
##   Type Alcohol Malic  Ash Alcalinity Magnesium Phenols Flavanoids
## 1    1   14.23  1.71 2.43       15.6       127    2.80       3.06
## 2    1   13.20  1.78 2.14       11.2       100    2.65       2.76
## 3    1   13.16  2.36 2.67       18.6       101    2.80       3.24
## 4    1   14.37  1.95 2.50       16.8       113    3.85       3.49
## 5    1   13.24  2.59 2.87       21.0       118    2.80       2.69
## 6    1   14.20  1.76 2.45       15.2       112    3.27       3.39
##   Nonflavanoids Proanthocyanins Color  Hue Dilution Proline
## 1          0.28            2.29  5.64 1.04     3.92    1065
## 2          0.26            1.28  4.38 1.05     3.40    1050
## 3          0.30            2.81  5.68 1.03     3.17    1185
## 4          0.24            2.18  7.80 0.86     3.45    1480
## 5          0.39            1.82  4.32 1.04     2.93     735
## 6          0.34            1.97  6.75 1.05     2.85    1450
```

```
# Exclude the "Type" variable from the data inputs
# and assign the others to data.train
data.train <- wine[-1]
# Use set seed() so that the clustering results can be reproducible
set.seed(2475)
# Use k-means method to cluster the wines into 3 groups
fit.km <- clusterByKmeans(data.train,3)
# Visualize the clustering results using plotcluster() from the fpc library
plotcluster(data.train, fit.km)
```



As can be seen from the above plot, there exist some overlaps between clusters, for instance, between cluster 1 and cluster 2. The data points are indeed separated by the K means but it may not be very suitable to say that the data are well-separated.

We can compare the clustering results by k-means and the original classification of the 178 data indicated by the variable “Type”:

```
# A comparison of the clustering results by k-means
# and the original classification indicated by "Type"
wt.km <- table(wine$Type, fit.km)
wt.km
```

```
##    fit.km
##      1  2  3
##   1  0 11 48
##   2 50 19  2
##   3 17 31  0
```

```
# Calculate the adjusted rand index, which is used for
## quantifying to what extent the two ways of clustering agree with each other.
randIndex(wt.km)
```

```
##       ARI
## 0.3986227
```

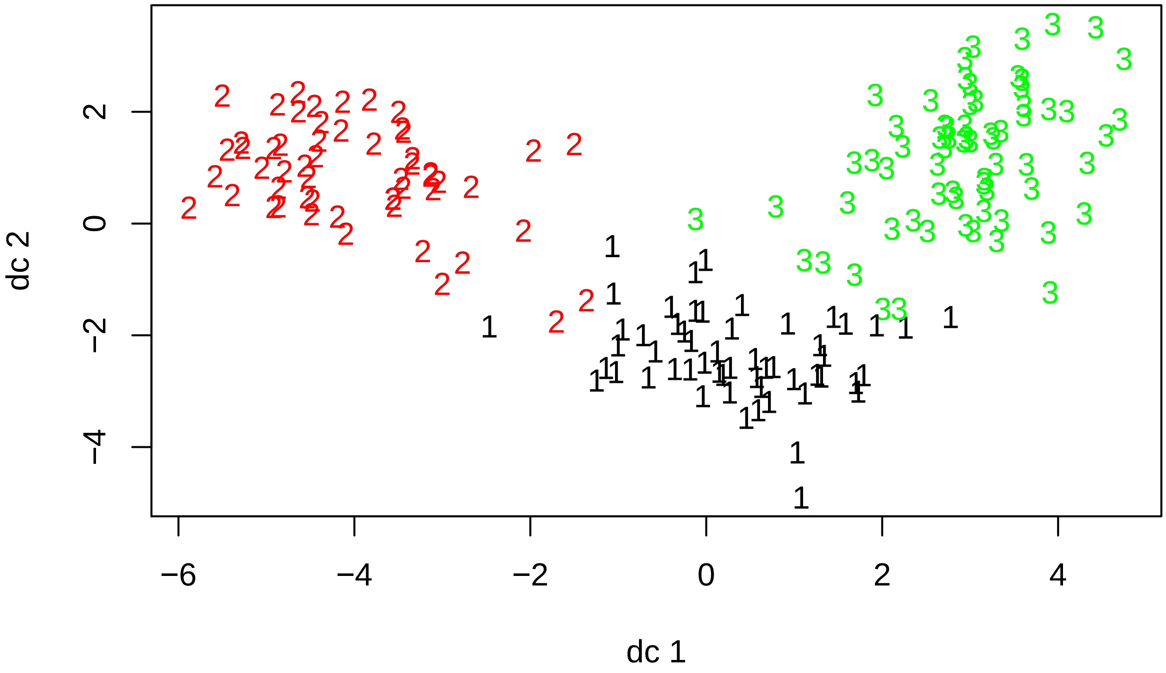
The above table shows directly that differences between the two partitions indeed exist. The value of the adjusted rand index is 0.3986227, which is not close to 1. Note that the adjusted rand index which is a measure of the degree of agreement between two partitions takes values between -1 (implying no agreement) and 1 (implying perfect agreement). Hence, we conclude that the partition obtained by K means does not match the original types of the wine well.

Now we repeat the above exercise using scaled data:

```
# Obtain the wine data from the package "rattle"
data(wine, package="rattle")
head(wine)
```

```
##   Type Alcohol Malic  Ash Alcalinity Magnesium Phenols Flavanoids
## 1    1   14.23  1.71 2.43       15.6       127    2.80       3.06
## 2    1   13.20  1.78 2.14       11.2       100    2.65       2.76
## 3    1   13.16  2.36 2.67       18.6       101    2.80       3.24
## 4    1   14.37  1.95 2.50       16.8       113    3.85       3.49
## 5    1   13.24  2.59 2.87       21.0       118    2.80       2.69
## 6    1   14.20  1.76 2.45       15.2       112    3.27       3.39
##   Nonflavanoids Proanthocyanins Color  Hue Dilution Proline
## 1          0.28            2.29  5.64 1.04     3.92    1065
## 2          0.26            1.28  4.38 1.05     3.40    1050
## 3          0.30            2.81  5.68 1.03     3.17    1185
## 4          0.24            2.18  7.80 0.86     3.45    1480
## 5          0.39            1.82  4.32 1.04     2.93     735
## 6          0.34            1.97  6.75 1.05     2.85    1450
```

```
# Exclude the "Type" variable from the data inputs,
# center and scale the rest of the data and assign the rest to data.train
data.train <- scale(wine[-1])
# Use set seed() so that the clustering results can be reproducible.
set.seed(2475)
# Use k-means method to cluster the wines into 3 groups
fit.km <- clusterByKmeans(data.train,3)
# Visualize the clustering results using plotcluster() from the fpc library
plotcluster(data.train, fit.km)
```



The command, data.train <- scale(wine[-1]), is used for center and scale the wine data excluding the variable Type. More specifically, for each column of wine[-1], first substract the corresponding column mean and then dividing that centered column by its standard deviation. In the data.train here, each column has a mean of zero.

Using the scaled data, we obtain a clustering result which looks better than what we obtained in the unscaled case, though minor overlaps still exist.

We then compare the clustering results by k-means and the original classification of the 178 data indicated by the variable "Type":

```
# A comparison of the clustering results by k-means
# and the original claassification indicated by "Type"
wt.km <- table(wine$Type, fit.km)
wt.km
```

```
##    fit.km
##      1  2  3
##   1  0  0 59
##   2 55  7  9
##   3  0 48  0
```

```r
# Calculate the adjusted rand index
randIndex(wt.km)
```

```
##       ARI
## 0.7423375
```

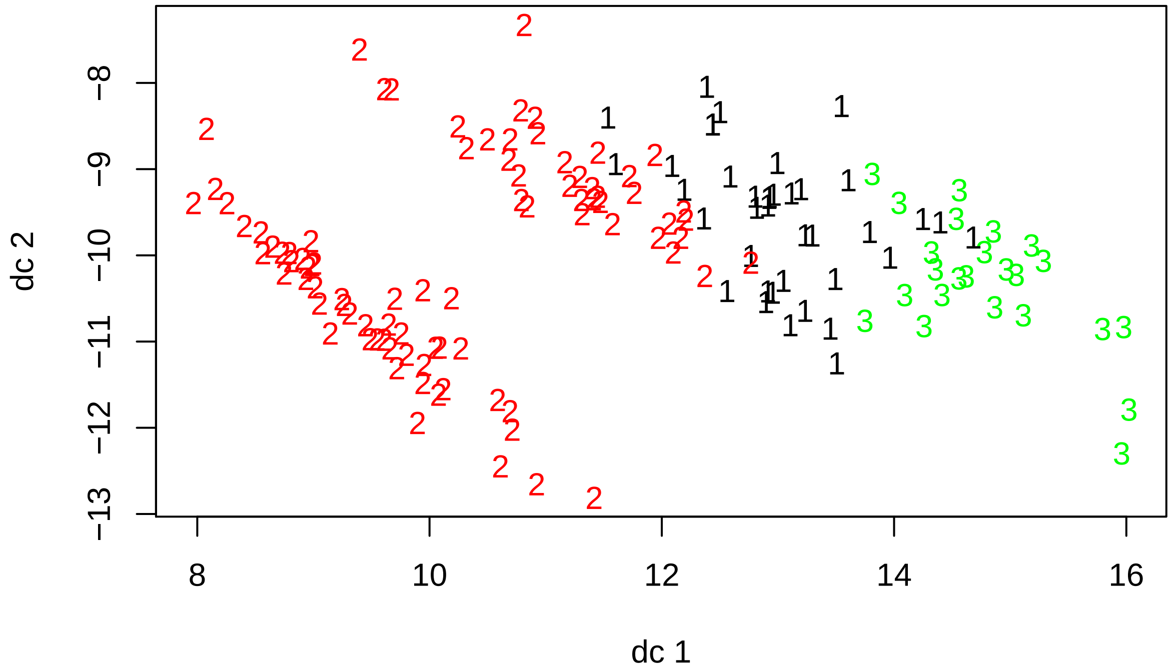
Now the table shows that there are fewer missing samples than in the unscaled case. The value of the adjusted rand index is 0.7423375, which is much larger than in the unscaled case. Hence, scaling helps improve the performance of K means in the sense that after scaling the clustering result using K means is more similar to the original partition.

In the following we show repeat the above exercise for the iris dataset. We first use the original dataset which is not scaled.

```r
# Obtain the iris data
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
# Exclude the "Species" variable and assign the rest to data.train
data.train <- iris[-5]
# Use set seed() so that the clustering results can be reproducible
set.seed(3000)
# Use k-means method to cluster the data into 3 groups
fit.km <- clusterByKmeans(data.train,3)
# Visualize the clustering results using plotcluster() from the fpc library
plotcluster(data.train, fit.km)
```



The plot here shows some overlaps between clusters, for instance, between cluster 1 and cluster 3 as well as between cluster 1 and cluster 2. It seems that K means does not separate the data points very well in this case.

Now we compare the clustering results by k-means and the original classification of the data indicated by the variable "species":

```
fs.km <- table(iris$Species,fit.km)
fs.km
```

```
##             fit.km
##               1  2  3
##   setosa      0 50  0
##   versicolor 14 36  0
##   virginica  25  1 24
```

```
randIndex(fs.km)
```

```
##       ARI
## 0.3609973
```

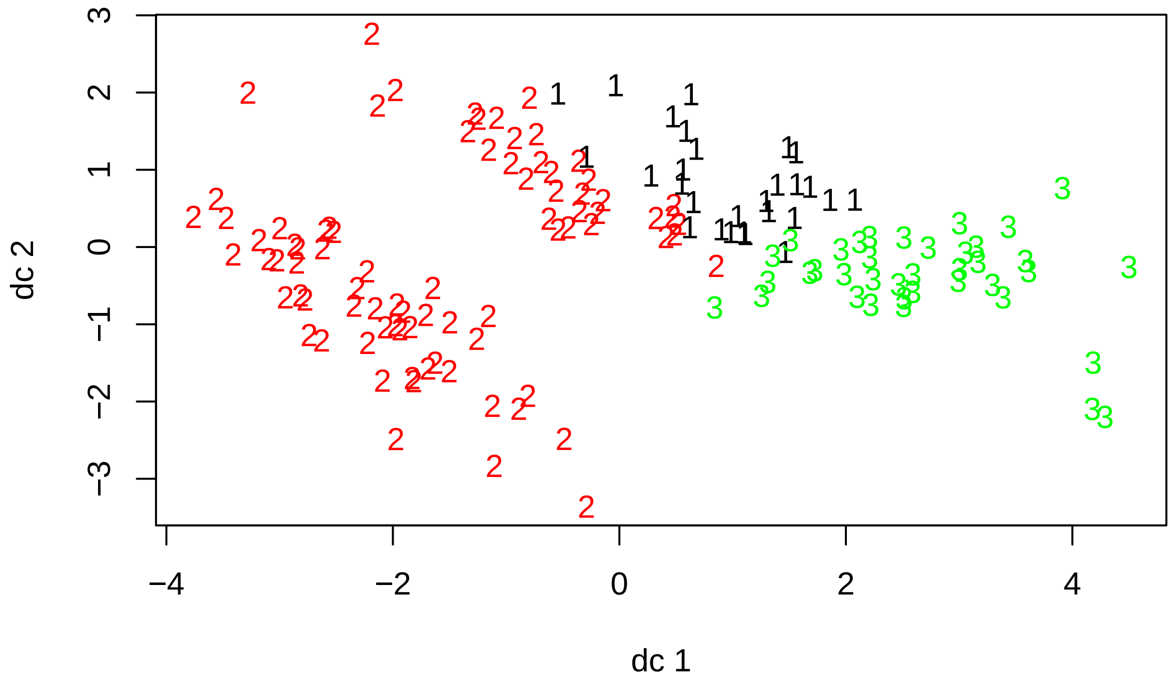
The adjusted rand index here is 0.3609973, which is not close to 1. So the algorithm's clusters do not match the original partition very well.

Now we instead use the scaled iris dataset:

```r
# Obtain the iris data
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
# Exclude the "Species" variable and assign the rest to data.train
data.train <- scale(iris[-5])
# Use set seed() so that the clustering results can be reproducible
set.seed(3000)
# Use k-means method to cluster the data into 3 groups
fit.km <- clusterByKmeans(data.train,3)
# Visualize the clustering results using plotcluster() from the fpc library
plotcluster(data.train, fit.km)
```



As shown in the plot, there are still some overlaps clusters, for instance, between cluster 1 and cluster 2 as well as between cluster 1 and cluster 3. K-means here still cannot separate the data points very well.

To quantify the comparison, we can still apply randIndex()

```
fs.km <- table(iris$Species,fit.km)
fs.km
```

```
##             fit.km
##               1  2  3
##   setosa      0 50  0
##   versicolor 10 32  8
##   virginica  19  1 30
```

```
randIndex(fs.km)
```

```
##       ARI
## 0.3510718
```

The adjusted rand index now is 0.3510718, which is even smaller than the unscaled case. In other words, for the iris dataset, scaling does not help improving the effectiveness of the k-means and K-means method in this case works better for the unscaled case.

As a summary, scaling is useful for improving the performance of K means for the wine data but not for the iris data.

Remark: when performing the K-means algorithm, we observe that the performance of this algorithm depends on the initial choice of group centers. Nevertheless, we choose the set seeds in this part so as to make the output reproducible.